Table of Contents

[Trees Are Not Strong on their Own 1](#_Toc55905048)

[Random Forests 1](#_Toc55905049)

[Advantage of Random Forests 2](#_Toc55905050)

[Disadvantage of Random Forests 2](#_Toc55905051)

[Building a Simple Random Forest Model 2](#_Toc55905052)

[Displaying Feature Importance 5](#_Toc55905053)

[Random Forest Hyperparameter Tuning 8](#_Toc55905054)

[Random forests have several hyperparameters: 8](#_Toc55905055)

## Trees Are Not Strong on their Own

Last week we looked at decision trees for regression. On their own they are not great. They may work well with training data but they are not good with new samples.

## Random Forests

Random forests combine hundreds of trees. These trees are built and initialized slightly differently. Once the random forest is built, the test data is given to each tree in the forest. Each tree makes a prediction. Each prediction counts as a vote for the collective outcome. Random forests can be used for classification and linear regressions.



Once again Josh Stormer provides a great video on now to build a random forest. I recommend watching it:

<https://www.youtube.com/watch?v=J4Wdy0Wc_xQ>

To create a forest:

* Each tree is created with a bootstrapped data set. “Bootstrapped” means the tree is built using sampling with replacement.
* Each decision tree is built with a random subset of variables at each step.
* Hundreds of trees are built.

To evaluate the forest, test data is given to the forest and the trees vote on the prediction.

### Advantage of Random Forests

Random forests include the following:

* They are often highly accurate.
* They help to avoid overfitting.
* They enable helpful scoring of important features so the least important features can be dropped during training.

### Disadvantage of Random Forests

The downside of random forests includes:

* Slowness.
* Difficult interpretability.

## Building a Simple Random Forest Model

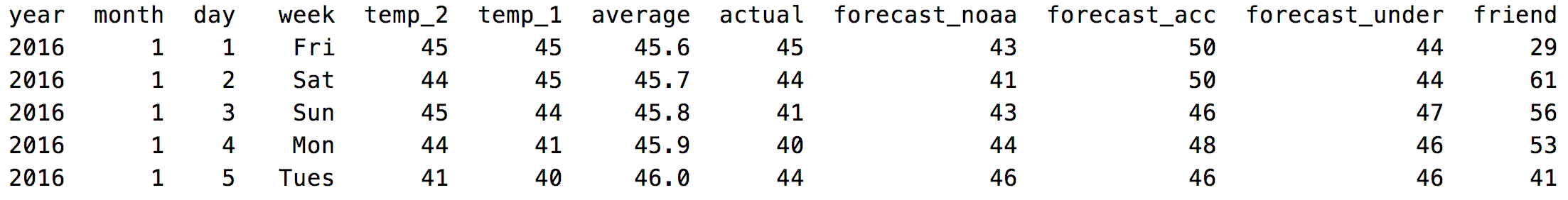
The code that builds a random forest is similar to many other models that we have used. In the snippet below, a random forest with 1000 trees is created. We will fine tune these parameters later.

|  |
| --- |
| # Import the model we are using  from sklearn.ensemble import RandomForestRegressor  # Instantiate model with 1000 decision trees  rf = RandomForestRegressor(n\_estimators = 1000, random\_state = 42)  # Train the model on training data  rf.fit(train\_features, train\_labels)  # Use the forest's predict method on the test data  predictions = rf.predict(test\_features) |

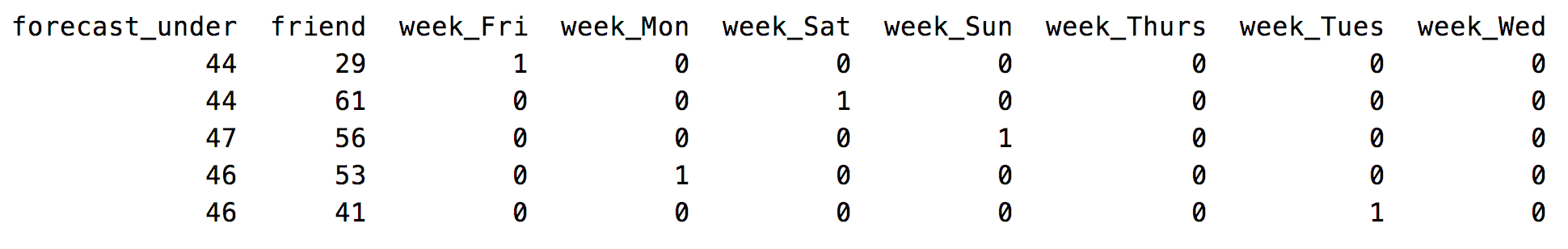
Example : Random Forests

This example implements a random forest to predict the weather. It is based on the code that is presented at <https://towardsdatascience.com/random-forest-in-python-24d0893d51c0>

Here is a quick view of the data set:



When preparing the data, one-hot encoding is used to create dummy variables for the days of the week:



The random forest produces a reasonably accurate model. The example provided implements the mean absolute error which is used to calculate an accuracy rating.

Mean Absolute Error =

# Calculate the absolute errors

errors = abs(predictions - test\_labels)

# Calculate mean absolute percentage error (MAPE)

mape = 100 \* (errors / test\_labels)

# Calculate and display accuracy

accuracy = 100 - np.mean(mape)

The root mean square error has also been included with the results as another valid measure of model success.

|  |
| --- |
| Mean Absolute Error: 3.87 degrees.  Accuracy: 93.93 %.  RMSE: 5.101657512937373 |

Here is the full program:

|  |
| --- |
| # Pandas is used for data manipulation  import pandas as pd  from sklearn.metrics import mean\_squared\_error  # Read in data and display first 5 rows  features = pd.read\_csv('/Users/pm/Desktop/DayDocs/data/temperatures.csv')  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(features)  # One-hot encode the data using pandas get\_dummies  features = pd.get\_dummies(features)  # Display the first 5 rows of the last 12 columns.  print(features.head(5))  # Use numpy to convert to arrays  import numpy as np  # Labels are the values we want to predict  labels = np.array(features['actual'])  # Remove the labels from the features  # axis 1 refers to the columns  features= features.drop('actual', axis = 1)  # Saving feature names for later use  feature\_list = list(features.columns)  # Convert to numpy array  features = np.array(features)  # Using Skicit-learn to split data into training and testing sets  from sklearn.model\_selection import train\_test\_split  # Split the data into training and testing sets  train\_features, test\_features, train\_labels, test\_labels =\  train\_test\_split(features, labels, test\_size = 0.25, random\_state = 42)  # Import the model we are using  from sklearn.ensemble import RandomForestRegressor  # Instantiate model with 1000 decision trees  rf = RandomForestRegressor(n\_estimators = 1000, random\_state = 42)  # Train the model on training data  rf.fit(train\_features, train\_labels)  # Use the forest's predict method on the test data  predictions = rf.predict(test\_features)  # Calculate the absolute errors  errors = abs(predictions - test\_labels)  # Print out the mean absolute error (mae)  print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')  # Calculate mean absolute percentage error (MAPE)  mape = 100 \* (errors / test\_labels)  # Calculate and display accuracy  accuracy = 100 - np.mean(mape)  print('Accuracy:', round(accuracy, 2), '%.')  # Print out the mean square error.  mse = mean\_squared\_error(test\_labels, predictions)  print('RMSE:', np.sqrt(mse)) |

## Displaying Feature Importance

A neat feature of random forests is its ability to provide a ranking of feature importance for making predictions.

Example : Show the Importance’s

This example demonstrates how to rank feature importance for the random forest. The output is:

|  |
| --- |
| Variable: temp\_1 Importance: 0.66  Variable: average Importance: 0.15  Variable: forecast\_noaa Importance: 0.05  Variable: forecast\_acc Importance: 0.03  Variable: day Importance: 0.02  Variable: temp\_2 Importance: 0.02  Variable: forecast\_under Importance: 0.02  Variable: friend Importance: 0.02  Variable: month Importance: 0.01  Variable: year Importance: 0.0  Variable: week\_Fri Importance: 0.0  Variable: week\_Mon Importance: 0.0  Variable: week\_Sat Importance: 0.0  Variable: week\_Sun Importance: 0.0  Variable: week\_Thurs Importance: 0.0  Variable: week\_Tues Importance: 0.0  Variable: week\_Wed Importance: 0.0 |

To implement this solution, add the following code to the end of Example 1.

|  |
| --- |
| # Get numerical feature importances  importances = list(rf.feature\_importances\_)  # Present features and importance scores.  def showFeatureImportances(importances, feature\_list):  dfImportance = pd.DataFrame()  for i in range(0, len(importances)):  dfImportance = dfImportance.append({"importance":importances[i],  "feature":feature\_list[i] },  ignore\_index = True)  dfImportance = dfImportance.sort\_values(by=['importance'],  ascending=False)  print(dfImportance)  showFeatureImportances(importances, feature\_list) |

Example : Building Forest with Important Features Only

This example builds a random forest with the important features which includes only the *temp1* variable and the *average* variable. The following scores are surprisingly competitive with the results from Example 2 which use many more features.

|  |
| --- |
| Mean Absolute Error: 3.92 degrees.  Accuracy: 93.76 %.  RMSE: 5.167988371104833 |

To build this solution, add this code to the end of Example 2:

|  |
| --- |
| # New random forest with only the two most important variables  rf\_most\_important = RandomForestRegressor(n\_estimators= 1000, random\_state=42)  # Extract the two most important features  important\_indices = [feature\_list.index('temp\_1'), feature\_list.index('average')]  train\_important = train\_features[:, important\_indices]  test\_important = test\_features[:, important\_indices]  # Train the random forest  rf\_most\_important.fit(train\_important, train\_labels)  # Make predictions and determine the error  predictions = rf\_most\_important.predict(test\_important)  errors = abs(predictions - test\_labels)  # Display the performance metrics  print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')  mape = np.mean(100 \* (errors / test\_labels))  accuracy = 100 - mape  print('Accuracy:', round(accuracy, 2), '%.') |

Exercise (4 marks)

Given the following code for the Iris data set, add the additional code needed to find the top three features for a random forest where. Hint: (See Example 3)

feature\_list = ['sepal length', 'sepal width', 'petal length', 'petal width']

Here is code to help you get started:

|  |
| --- |
| #Import scikit-learn dataset library  from sklearn import datasets  #Load dataset  iris = datasets.load\_iris()  # Creating a DataFrame of given iris dataset.  import pandas as pd  data=pd.DataFrame({  'sepal length':iris.data[:,0],  'sepal width':iris.data[:,1],  'petal length':iris.data[:,2],  'petal width':iris.data[:,3],  'species':iris.target  })  iris['target\_names']  print(data.head())  # Import train\_test\_split function  from sklearn.model\_selection import train\_test\_split  X=data[['sepal length', 'sepal width', 'petal length', 'petal width']] # Features  y=data['species'] # Labels  # Split dataset into training set and test set  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)  #Import Random Forest Model  from sklearn.ensemble import RandomForestClassifier  #Create a Gaussian Classifier  rf=RandomForestClassifier(n\_estimators=100)  #Train the model using the training sets y\_pred=rf.predict(X\_test)  rf.fit(X\_train,y\_train)  y\_pred=rf.predict(X\_test)  #Import scikit-learn metrics module for accuracy calculation  from sklearn import metrics  # Model Accuracy, how often is the classifier correct?  print("Accuracy:", metrics.accuracy\_score(y\_test, y\_pred))  # Predict species for a single flower.  # sepal length = 3, sepal width = 5  # petal length = 4, petal width = 2  prediction = rf.predict([[3, 5, 4, 2]])  # 'setosa', 'versicolor', 'virginica'  print(prediction) |

Add in code to generate the top performing feature list. Show the listing that is displayed when running your revised program here:

|  |
| --- |
| #Import scikit-learn dataset library from sklearn import datasets import pandas as pd from sklearn.metrics import mean\_squared\_error   #Load dataset iris = datasets.load\_iris()  # Creating a DataFrame of given iris dataset. import pandas as pd data=pd.DataFrame({  'sepal length':iris.data[:,0],  'sepal width':iris.data[:,1],  'petal length':iris.data[:,2],  'petal width':iris.data[:,3],  'species':iris.target }) iris['target\_names'] print(data.head())  # Import train\_test\_split function from sklearn.model\_selection import train\_test\_split X=data[['sepal length', 'sepal width', 'petal length', 'petal width']] # Features y=data['species'] # Labels  # Split dataset into training set and test set X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)  #Import Random Forest Model from sklearn.ensemble import RandomForestClassifier  #Create a Gaussian Classifier rf=RandomForestClassifier(n\_estimators=100)  #Train the model using the training sets y\_pred=rf.predict(X\_test) rf.fit(X\_train,y\_train)  y\_pred=rf.predict(X\_test)  #Import scikit-learn metrics module for accuracy calculation from sklearn import metrics # Model Accuracy, how often is the classifier correct? print("Accuracy:", metrics.accuracy\_score(y\_test, y\_pred))  # Predict species for a single flower. # sepal length = 3, sepal width = 5 # petal length = 4, petal width = 2 prediction = rf.predict([[3, 5, 4, 2]]) # 'setosa', 'versicolor', 'virginica' print(prediction)    # Show all columns. pd.set\_option('display.max\_columns', None) pd.set\_option('display.width', 1000)   # One-hot encode the data using pandas get\_dummies # features = pd.get\_dummies(features) features = pd.get\_dummies(data) print(features)  # Display the first 5 rows of the last 12 columns. print(features.head(5))  # Use numpy to convert to arrays import numpy as np  # Labels are the values we want to predict labels = np.array(features['species'])  # Remove the labels from the features # axis 1 refers to the columns features= features.drop('species', axis = 1)  # Saving feature names for later use feature\_list = list(features.columns)  # Convert to numpy array features = np.array(features)  # Using Skicit-learn to split data into training and testing sets from sklearn.model\_selection import train\_test\_split  # Split the data into training and testing sets train\_features, test\_features, train\_labels, test\_labels =\  train\_test\_split(features, labels, test\_size = 0.25, random\_state = 42)  # Import the model we are using from sklearn.ensemble import RandomForestRegressor  # Instantiate model with 1000 decision trees rf = RandomForestRegressor(n\_estimators = 1000, random\_state = 42)  # Train the model on training data rf.fit(train\_features, train\_labels)  # Use the forest's predict method on the test data predictions = rf.predict(test\_features)  # Calculate the absolute errors errors = abs(predictions - test\_labels)  # Print out the mean absolute error (mae) print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.')  # Calculate mean absolute percentage error (MAPE) mape = 100 \* (errors / test\_labels)  # Calculate and display accuracy accuracy = 100 - np.mean(mape)  print('Accuracy:', round(accuracy, 2), '%.')  # Print out the mean square error. mse = mean\_squared\_error(test\_labels, predictions) print('RMSE:', np.sqrt(mse))  # Get numerical feature importances importances = list(rf.feature\_importances\_)  # Present features and importance scores. def showFeatureImportances(importances, feature\_list):  dfImportance = pd.DataFrame()  df\_list =[ ]  for i in range(0, len(importances)):  new\_row = {"importance": importances[i], "feature": feature\_list[i]}  df\_list.append({"importance": importances[i], "feature": feature\_list[i]})  print(new\_row)  # dfImportance = pd.concat((dfImportance, df\_new\_row))  # print(df\_list)  df\_list = pd.DataFrame(df\_list)  dfImportance = pd.concat(df\_list, ignore\_index=True)    # dfImportance = pd.DataFrame().concat({"importance":importances[i],  # "feature":feature\_list[i] },  # ignore\_index = True)   dfImportance = dfImportance.sort\_values(by=['importance'],  ascending=False)  print(dfImportance)  print("#######importnace###########") showFeatureImportances(importances, feature\_list)  # New random forest with only the two most important variables rf\_most\_important = RandomForestRegressor(n\_estimators= 1000, random\_state=42)  # Extract the two most important features important\_indices = [feature\_list.index('petal length'), feature\_list.index('petal width')] train\_important = train\_features[:, important\_indices] test\_important = test\_features[:, important\_indices]  # Train the random forest rf\_most\_important.fit(train\_important, train\_labels)  # Make predictions and determine the error predictions = rf\_most\_important.predict(test\_important) errors = abs(predictions - test\_labels)  # Display the performance metrics print('Mean Absolute Error:', round(np.mean(errors), 2), 'degrees.') mape = np.mean(100 \* (errors / test\_labels)) accuracy = 100 - mape print('Accuracy:', round(accuracy, 2), '%.') |

Exercise (3 marks)

Rebuild the random forest with Iris data set using only the top performing features. Show the accuracy score. Compare the accuracy of the Iris data set before and after the removal of the lowest performing feature.

|  |
| --- |
| Text  Description automatically generated |

## Random Forest Hyperparameter Tuning

## Random forests have several hyperparameters:

* n\_estimators = number of trees in the forest.
* max\_features = max number of features considered for splitting a node. Settings include:
  + int, then consider max\_features features at each split.
  + float, then max\_features is a fraction and int(max\_features \* n\_features) features are considered at each split.
  + “auto”, then max\_features=sqrt(n\_features).
  + “sqrt”, then max\_features=sqrt(n\_features) (same as “auto”).
  + “log2”, then max\_features=log2(n\_features).
  + None, then max\_features=n\_features.
* max\_depth = max number of levels in each decision tree.
* min\_samples\_split = min number of data points placed in a node before the node is split.
* min\_samples\_leaf = min number of data points allowed in a leaf node.
* bootstrap = method for sampling data points (with or without replacement).

It is possible to use both RandomizedSearchCV() and GridSearchCV() to help find optimal parameters.

Example : Searching for the Optimal RandomForest Hyperparmaters

This example shows how to search for an optimal set of parameters with RandomizedSearchCV() for the temperatures.csv data set. When running the program with the weather set the output is:

{'n\_estimators': 1600, 'min\_samples\_split': 10, 'min\_samples\_leaf': 2, 'max\_features': 'auto', 'max\_depth': None, 'bootstrap': True}

To build this example, add this code to the end of Example 1.

|  |
| --- |
| from sklearn.model\_selection import RandomizedSearchCV  # Number of trees in random forest  n\_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]  # Number of features to consider at every split  max\_features = ['auto', 'sqrt']  # Maximum number of levels in tree  max\_depth = [int(x) for x in np.linspace(10, 110, num = 11)]  max\_depth.append(None)  # Minimum number of samples required to split a node  min\_samples\_split = [2, 5, 10]  # Minimum number of samples required at each leaf node  min\_samples\_leaf = [1, 2, 4]  # Method of selecting samples for training each tree  bootstrap = [True, False]  # Create the random grid  random\_grid = {'n\_estimators': n\_estimators,  'max\_features': max\_features,  'max\_depth': max\_depth,  'min\_samples\_split': min\_samples\_split,  'min\_samples\_leaf': min\_samples\_leaf,  'bootstrap': bootstrap}  {'bootstrap': [True, False],  'max\_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None],  'max\_features': ['auto', 'sqrt'],  'min\_samples\_leaf': [1, 2, 4],  'min\_samples\_split': [2, 5, 10],  'n\_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}  print(random\_grid)  # Use the random grid to search for best hyperparameters  # First create the base model to tune  rf = RandomForestRegressor()  # Random search of parameters, using 3 fold cross validation,  # search across 100 different combinations, and use all available cores  rf\_random = RandomizedSearchCV(estimator = rf, param\_distributions = random\_grid, n\_iter = 100, cv = 3, verbose=2, random\_state=42, n\_jobs = -1)  # Fit the random search model  rf\_random.fit(train\_features, train\_labels)  print("Best parrameters")  print(rf\_random.best\_params\_) |

Exercise (4 marks)

Using the randomized search parameter values that are listed before the code in Example 4, take the code from Example 1 and assign these parameter values when declaring the RandomForestRegressor() object. Show the new MAE and RMSE measures here after making the changes:

|  |
| --- |
| After  Text  Description automatically generated |

Compare these scores with the results that are obtained initially in Example 1:

|  |
| --- |
| Before  Text  Description automatically generated  After using Randomized searching parameter values, MAE is decreased, accuracy is increased and RMSE is decreased so the result are optimal |

Exercise (3 marks)

The following code performs logistic regression using Logistic Regression.

|  |
| --- |
| from sklearn import datasets  iris = datasets.load\_iris()  # Creating a DataFrame of given iris dataset.  import pandas as pd  data=pd.DataFrame({  'sepal length':iris.data[:,0],  'sepal width':iris.data[:,1],  'petal length':iris.data[:,2],  'petal width':iris.data[:,3],  'species':iris.target  })  iris['target\_names']  print(data.head())  # Import train\_test\_split function  from sklearn.model\_selection import train\_test\_split  X=data[['sepal length', 'sepal width', 'petal length', 'petal width']] # Features  y=data['species'] # Labels  # Split dataset into training set and test set  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)  from sklearn.preprocessing import StandardScaler  sc\_x = StandardScaler()  X\_train\_scaled = sc\_x.fit\_transform(X\_train)  X\_test\_scaled = sc\_x.transform(X\_test)  from sklearn import metrics  from sklearn.ensemble import RandomForestClassifier  from sklearn.linear\_model import LogisticRegression  def buildModelAndPredict(clf, X\_train\_scaled, X\_test\_scaled, y\_train, y\_test, title):  print("\n\*\*\*\* " + title)  #Train the model using the training sets y\_pred=rf.predict(X\_test)  clf\_fit = clf.fit(X\_train\_scaled,y\_train)  y\_pred = clf\_fit.predict(X\_test\_scaled)  print("Accuracy:", metrics.accuracy\_score(y\_test, y\_pred))  # For explanation see:  # https://towardsdatascience.com/multi-class-metrics-made-simple-part-i-precision-and-recall-9250280bddc2  print(metrics.classification\_report(y\_test, y\_pred, digits=3))  # Predict species for a single flower.  # sepal length = 3, sepal width = 5  # petal length = 4, petal width = 2  prediction = clf\_fit.predict([[3, 5, 4, 2]])  # 'setosa', 'versicolor', 'virginica'  print(prediction)  lr = LogisticRegression(fit\_intercept=True, solver='liblinear')  buildModelAndPredict(lr, X\_train\_scaled, X\_test\_scaled, y\_train, y\_test, "Logistic Regression") |

Modify the code to initialize a RandomForest classifier using 200 estimators and max\_features of 3. Show a comparison of the precision, recall and F1 scores here for both logistic regression and random forests.

|  |
| --- |
| Calendar  Description automatically generated |